COMPUTER SYSTEM with REAL-TIME COMPILATION

FIELD OF THE INVENTION

This invention relates to computer systems, and more particularly, to a novel computer architecture providing real-time compilation of a high-level language source program concurrently as the program is being entered or edited at the console by the programmer.

PRIOR APPLICATION

This application is a continuation-in-part of my prior copending application Serial No. 06/425,612, filed September 28, 1982, and having the same title. The specification and drawings of the present application are intended to be identical to said prior application, with the addition of an appendix containing the Pascal source code implementing the invention for Pascal-S and the IBM Personal Computer.

BACKGROUND OF THE INVENTION

Since humans write programs in a programming language and computers execute only machine language, it is frequently necessary to translate from one language to the other. When the programming language is "high level", that is, abstract in the sense that it does not explicitly manipulate the computer registers and other hardware, the translation of the original program is performed by another program called a "compiler". The orig-

inal program is called the "source code", and the resulting program translation is called the "object code".

In addition to translation, the compiler must also perform lexical, syntactic and semantic analyses of the source code. Lexical analysis is performed by a "scanner" and is the process of forming a sequence of source code bytes into meaningful symbols or tokens, somewhat like forming a sequence of characters into English words. These symbols are then subjected to the syntactic analysis by a "parser" which determines if they are arranged in a relation which conforms to the rigid grammatical rules of the programming language. The semantic analysis determines if the symbols conform to additional rules which cannot be conveniently expressed by the language grammar.

These analyses are very much like parsing the words of an English sentence. If the sequence of symbols violates a syntactic or semantic rule an "error" is said to have been committed and the compiler must so inform the programmer by emitting a visible "error message".

After translation the resulting object code is usually "linked" and "loaded", processes in which it is joined with other object code modules to form a complete machine code program which may be executed by the computer.

DESCRIPTION OF THE PRIOR ART

In recent years the large increase in software costs, the lack of skilled programmers, the rapid expansion of the computer market, the widespread adoption of microcomputers, and the underutilization of much available hardware because of lack of software, have compelled the adoption of high-level languages and concerted efforts to make their use more efficient.

However, programming in a high-level language is still slow, tedious and inefficient. For example, even under the optimum conditions of an interactive console, a compiled language requires a repeated sequence of steps comprising loading the editor, writing or editing the source code, loading the compiler, executing the compiler, loading the linker, executing the linker, running the program, and repeating the sequence when an error is indicated during compilation of the source code or execution of the object code. During much of the time the programmer is compelled to wait for completion of the loading or execution steps, and this waiting time is both wasteful and boring. As a result, the programming process is slow and expensive. It is generally accepted that the output of the average professional programmer is only about five to ten lines of debugged source code per day (Ref. 4). The parenthetical references are to the Bibliography at the end of this specification.

The recent widespread use of personal microcomputers has compounded the programming problem. Most users presently writing programs for such computers are not professionally trained as programmers and are unwilling to expend the time and effort or to endure the tedium and frustration inherent in the mechanics of programming in a compiled language with present microcomputer systems (Ref. 9).

Instead, the majority of personal computer programmers utilize BASIC interpreters. The latter are generally syntactically sparse, ineffectual in revealing errors, incapable of utilizing local variables, unable to pass parameters to subroutines, unable to invoke subroutines by name, incapable of linking library modules, and lacking in both data structures and flow control structures, so as to make it extremely difficult to

write error-free programs for any but the simplest applications. BASIC is therefore regarded as "a very poor vehicle for teaching good program technique" (Ref. 1). These BASIC interpreters are also so slow in execution as to be disadvantageous or even useless for many applications.

Nevertheless, the mechanics of compilation with present microcomputer systems are so inconvenient that BASIC interpreters predominate the microcomputer field and substantially degrade the programming process.

Numerous attempts have been made to minimize the disadvantages of conventional compiler usage. One such scheme is the so-called "incremental compiler" (Refs. 5, 6, 7, 10, 11). As each line of source code is entered at the console it is analyzed for conformity with the syntax rules of a local limited grammar, in which the line is treated in isolation without consideration of the context of the entire program. If the line is error-free from this limited consideration the programmer is free to enter the next line of code. Otherwise an error message is displayed and the error must be corrected before further lines are entered. After the entire program is entered the program undergoes further syntactic and semantic analyses with respect to the context-dependent rules, after which code generation and execution may take place.

The incremental compiler scheme has some merit when used with those languages which have few context-dependent restraints, such as BASIC. For modern structured languages such as Algol, Pascal, PL/I, C and Ada the limited local analysis which can be performed after entry of each line is only a relatively small portion of the total analysis required and not worth the

overhead.

Another well-known prior art scheme has been referred to as "repeated recompilation" (Ref. 2). In this approach the syntactic and semantic analyses after entry of each line include all context-dependent rules and consider the entirety of the partial program entered to that point. Therefore, upon editing of even a single byte of the entered source code, the entire source must be recompiled from the beginning. Since this recompilation must be completed before new lines may be entered, the large overhead of this scheme prohibits its use with any but the smallest programs.

In an effort to avoid the overhead of complete recompilation, a number of schemes of partial recompilation have been devised (Refs. 1, 3, 8). These schemes generally involve a complex data structure for the source code, and/or a complex editor, whereby a programmer having sufficient expertise in the particular scheme may make a change in the source code with the requirement of only a recompilation of the changed portion of the source code. These approaches require an excessive amount of computer memory and/or highly specialized skill of the programmer.

These prior art schemes have had no use except as experimental curiosities in academic research. Commercially available compilers still require the repeated sequences of the load, edit, load, compile, load, link and run steps as described above.

OBJECT OF THE INVENTION

The ideal computer architecture for the programming function

should have the following characteristics:

The compilation should be performed in real-time as the source code is entered or edited by the programmer;

The system should be capable of implementing a modern block-structured language having a powerful syntax, such as Algol, C, Pascal or PL/I;

There should be no waiting for disk accesses during program writing, editing or compilation, except to the extent necessary for initial entry or saving of source files;

An error message should be displayed substantially instantly after a syntax error is entered at the console;

Correction of source code errors should be fast and easy, without reloading the editor, source file or compiler;

Compilation should be finished almost instantly after the last line of an error-free source program is entered;

The compilation should be complete in that no further syntactic or semantic analysis is required, and the absence of an error message shold assure that the program has no syntactic or semantic errors;

The programmer should be able to stop execution of the object code at any time, examine the values of all variables, and continue execution, all without requiring the prior insertion of write statements, breakpoints or other debugging code into the source program; and

The architecture should not impose any additional requirements of unique language syntax, complex editor mechanics or scheme-specialized programmer expertise.

It is the primary object of the present invention to provide a novel computer system which closely approaches this ideal architecture.

SUMMARY OF THE INVENTION

The following is a summary description of a preferred embodiment of the invention.

The programmer invokes the real-time compiler-editor system by typing its command file name at the keyboard console. The command file containing the software portion of the system is then read into memory from a disk. The source buffer, a memory region which is to contain the source code, is initialized so that its first stored byte is a predetermined code which will be called a "Pause Mark". Execution of the compiler then begins by reading the Pause Mark as the character in the first location of the buffer. When the compiler reads the Pause Mark it will enter an infinite loop repeatedly reading the same location until the content of this location is changed by the editor to a blank (space).

When the programmer strikes a key on the console keyboard the central processor unit executes the following interrupt sequence described for an 8080 or Z80 microprocesor: Upon completion of the instruction currently being executed the processor will enter the interrupt mode and communicate its new status to the system by emitting an interrupt acknowledge signal.

Upon receipt of this signal the interrupt hardware gates an RST instruction onto the data bus. The processor then executes the RST instruction which is a one-byte call to a selected location in low memory where there is stored a jump instruction ("vector") to the interrupt service routine comprising the editor.

The interrupt service routine first saves the stack pointer and other CPU registers. If the struck key corresponds to an

alphanumeric or other non-control character the latter is placed into the second location of the source code buffer immediately after the Pause Mark. The buffer pointer is then advanced to the next location, the CPU registers are restored, the CPU interrupt enabled, and the RET (return) instruction executed to return control to the compiler.

The compiler continues to execute its infinite loop in which it repeatedly reads the Pause Mark character in the first location of the source code buffer. This sequence is repeated as the programmer strikes additional keys at the keyboard, the successive characters being entered into successive locations in the source code buffer as the buffer pointer advances. This sequence continues until a key corresponding to a control character is struck.

If this control character is a carriage return the corresponding code (13) is inserted into the buffer, the buffer pointer is advanced, the Pause Mark code is then inserted into the buffer location adjacent the carriage return code, and the original Pause Mark code in the first location is replaced by the code (32) for a blank space. The Pause Mark location has thus been advanced from its original point to the end of the first line of the source code.

Upon return to the compiler from the interrupt service routine the compiler pointer accesses the first buffer location and reads the code for a space instead of the Pause Mark. The compiler will then repeatedly advance its pointer to the next buffer location and perform its lexical, syntactic and semantic analyses on the first line of source code stored in the buffer. The compiler may either display an error message or emit compiled object code, as may be appropriate, until the compiler pointer

reaches the new Pause Mark inserted at the end of the first line of source code.

When it reaches the new Pause Mark the compiler again enters an infinite loop without advancing the pointer until the editor eventually moves the Pause Mark to the end of the next line, whereupon the compiler is free to compile this next line of source code.

Control characters other than a carriage return may be entered by striking appropriate keys to perform the conventional editing functions of a screen editor. For example, errors in the present line of source code may be corrected by moving the cursor backward. This does not affect the compiler which cannot advance beyond the Pause Mark at the end of the previous line.

However, if by hitting the appropriate control key the cursor is moved upwardly one or more lines to a position before the Pause Mark this sets a Recompile Flag so as to enter a recompile mode. In this event, upon return to the compiler the latter is reinitialized so that it may recompile the source code from the very beginning of the source buffer. Subsequent editing or source text insertions cause the editor to move the Pause Mark to an updated location adjacent the end of the line preceding the most recently edited line.

When the compiler finds a syntax error in the source code it displays an error message. The programmer may then edit the source so as to correct the error. Upon return from the editor the compiler is reinitialized to recompile the source code.

The entered source code and emitted object code are preferably stored in memory so that disk accesses will not unduly interfere with the editing and compilation processes. If the source or object code buffer gets filled its contents may be

stored in a disk file in the conventional manner as employed by editors and word processors such as CP/M ED and WordStar. The use of bank-select memory schemes or the advent of 16-bit microprocessors with their larger memory space will obviate the need for disk storage until the compilation is finished.

DESCRIPTION OF THE DRAWINGS

Fig. 1 is a schematic diagram showing the relation of the major hardware components constituting a preferred embodiment of the computer system in accordance with the present invention;

Fig. 2 is a diagram showing the interrupt logic and circuitry of the system hardware;

Fig. 3 is a flowchart showing the sequence of operations;

Fig. 4 is a flowchart showing the sequence of operations of the compiler;

Fig. 5 is a flowchart showing the sequence of operations of the editor; and

Fig. 6 is a flowchart showing the sequence of operations of the control-character routines of the editor.

DETAILED DESCRIPTION

The following is a detailed description of a preferred embodiment of the invention. The disclosed details are merely illustrative of one of the many forms which the invention may take in practise. The invention and novelty reside in neither the hardware nor the software taken separately, but rather in the novel combination of both.

Referring first to Fig. 1, there are shown the major hardware components constituting the overall system of a preferred embodiment of the present invention. Each component will be

referred to by the legend shown in the respective rectangle of the drawing. The CRT CONSOLE refers to any suitable terminal having a keyboard for entry of the source code to be compiled and also for entry of editing commands to change the code. The terminal also comprises a video display for implementation of a screen editor. The keyboard is preferably integral with the video display to form a unitary console having an RS-232-C serial link to the remainder of the system.

This serial link is connected to the INPUT PORT which is preferably embodied as a UART (univeral asynchronous receiver transmitter) such as, for example, the 1602, AY-5-1013, or TMS 5501. Each keystroke on the keyboard of the CRT CONSOLE results in the serial transmission to the UART of a train of bits constituting the ASCII byte corresponding to the struck key. The UART reforms the bits into that byte which is then transmitted in parallel on the data bus to the accumulator of the CPU (central processor unit). The UART also provides an output port. Execution of an OUT command by the CPU results in the transmission on the data bus of a byte from the accumulator to the UART which may then serially transmit the byte to the CRT CONSOLE for display on the video screen.

In the usual operating mode of a conventional microcomputer system the status of the input port is repeatedly tested by the central processor unit in a polling loop until the input port status indicates that a byte of data has been received and is available in the UART received data register. The present invention employs instead an interrupt mode of operation whereby the CPU normally executes the compiler until the UART receives a byte from the CRT CONSOLE. The compiler is stored in an area of main memory designated in FIG. 1 as COMPILER.

The data available line of the UART is then activated and this in turn activates the INTERRUPT CONTROLLER to cause the CPU to execute the editor. The latter is stored in an area of main memory designated in the drawing as EDITOR. Upon entry of the received character into the SOURCE BUFFER in main memory, or upon completion of an editing command, a RET (return) instruction is executed by the CPU to cause it to resume its execution of the COMPILER from the point where it was interrupted.

As the COMPILER is executed it preferably performs lexical, syntactic and semantic analyses of the program source code stored in the SOURCE BUFFER. In the preferred embodiment the COMPILER also emits object code and stores it in the OBJECT BUFFER. Upon completion of entry and compilation of the source code program, control of the CPU may be passed to the INTERPRETER for execution of the object code if the latter is in the form of intermediate code. The programmer may be given the option of saving the source code and/or object code in secondary storage such as disk or tape media. Instead of generating intermediate code ("p-code") for interpretation, the compiler may be of the type that emits executable machine code. The COMPILER may require only a single pass through the source code, in the manner of the usual recursive descent Pascal compiler. If the COMPILER requires more than one pass the first pass should preferably perform the syntax analysis so as to reveal all syntax errors.

The interrupt facility enables the programmer to stop execution of the machine code program at any time, examine the values of the variables, and then continue execution. No additional hardware is required for this extra function, and the extra software is minimal.

Referring now to Fig. 2, there are shown the circuitry and hardware components directly involved in the interrupt operation. Upon striking a key of the CRT CONSOLE a train of pulses constituting the byte corresponding to the struck key is emitted from the RS-232-C serial port O. A converter gate C1 converts the pulse train from RS-232-C levels to TTL (transistor-transistor-logic) levels to match the requirements of input port IN of the UART. The latter forms the serial pulse train into an eight-bit byte which is stored in the received data register of the UART. The latter then outputs a data available signal at pin DAV which signal is transmitted by gate G1 to a vectored interrupt input VI of the PRIORITY ENCODER.

Although only one input pin VI of the latter is shown, it will be understood that this chip has other vectored interrupt input pins to which other interrupting devices may be connected. The PRIORITY ENCODER arbitrates competing interrupt requests at its inputs and determines the request having the highest priority. The enable input EI of the PRIORITY ENCODER is grounded as shown.

Assuming that the interrupt request from the CONSOLE and the UART win the priority contest, the ENCODER then transmits a three-bit code A0,A1,A2 to the respective inputs of the INTERRUPT VECTOR REGISTER. The other five inputs of the latter are held positive by potential source +V, so that the resulting byte input to this register chip constitutes an RST call instruction. The signal at output pin GS of the PRIORITY ENCODER is transmitted by gate G2 to the latch enable input LE of the INTERRUPT VECTOR REGISTER to cause the latter to latch the RST call instruction into its internal flip-flops.

Activation of output pin GS of the PRIORITY ENCODER also

transmits an interrupt signal through AND gate A1 to the interrupt request pin INT* of the Z80 CPU. Assuming that the interrupt of the processor is enabled, upon completion of the present instruction the CPU's status pins IORQ* and M1* are activated and their signals are transmitted by gates G3,G4 to AND gate A2 to form the INTA(interrupt acknowledge) signal. The latter is inverted by gate G5 and fed to the output enable pin OE of the INTERRUPT VECTOR REGISTER, whereupon the RST call instruction at the inputs of the latter is jammed onto the DATA BUS.

The RST instruction is then input to and executed by the Z80 CPU, causing the latter to push the contents of the program counter onto the stack, and further causing the CPU to jump to a predetermined location in low memory. This location stores a "vector" or three-byte JMP (jump) instruction to an interrupt service routine. The latter includes the editor as well as a subroutine to store the contents of the CPU registers. Control of the CPU is then retained by the editor until either a character has been entered into the source code buffer or an editing operation has been completed.

The editor includes an input instruction which when executed causes the CPU to place the address of the UART's port on the ADDRESS BUS. This address is tested by the COMPARATOR, and if it matches that of the port, the output pin EOUT is activated to signal the DECODER. The latter is controlled by other control and status signals (not shown) in the conventional manner so as to transmit a signal RDE* to the corresponding input RDE of the UART. The byte in the received data register (not shown) of the UART is then gated onto the DATA BUS and transmitted to the accumulator within the CPU.

In the preferred embodiment of the invention shown in Fig. 2 the integrated circuits may be implemented as follows:

UART	: 1602
PRIORITY ENCODER	: 74LS148
INTERRUPT VECTOR REGISTER	: 74LS373
DECODER	: 74LS155
COMPARATOR	: 25LS2521
C1	: 1489
C2	: 1488

Referring now to Fig. 3 there is shown the sequence of operations of the overall system. The COMPILER normally has control of the CPU and either is in an infinite loop upon reaching a Pause Mark in the source code buffer or is in the process of analysing the source code in the buffer.

The occurrence of a KEYSTROKE at the terminal causes an INTERRUPT, whereupon the CPU is vectored to the interrupt service routine. The latter includes a subroutine to perform the SAVE REGISTERS procedure shown in the drawing.

The EDITOR is then executed by the CPU. If the KEYSTROKE corresponds to a control character, then an editing procedure such as a cursor movement, screen scroll, character deletion, or line deletion is performed. If the KEYSTROKE corresponds to an alphanumeric character or other valid source code character the latter is entered into the source code buffer and displayed on the video screen, and the screen cursor is advanced to the next character position.

The interrupt service routine then jumps to its subroutine to perform the RESTORE REGISTERS procedure whereby the registers

of the CPU are restored to their original values at the instant of the interrupt.

The ENABLE INTERRUPT instruction (EI) is then executed by the CPU so that the latter may respond to the next interrupt. Finally the RET instruction is executed so that the CPU may RETURN TO COMPILER. The compiler then resumes execution from the point where it was interrupted.

Referring now to Fig. 4, there is shown the sequence of operations of the compiler. After initialization, the compiler first performs its READ CHARACTER function whereby the byte in the first location of the source code buffer is read.

If this byte is the predetermined code designated as the Pause Mark the compiler pointer does not advance and the compiler enters an infinite loop wherein it continues to read the same location until the content of this location is changed by the editor to a blank. When this change occurs the compiler memory pointer is incremented to the next location of the buffer so that the compiler exits from its Pause loop, as indicated by the legend ADVANCE MEMORY POINTER.

As indicated by SYMBOL?, the lexical analyser of the compiler then determines if the source character read in constitutes the last character of a symbol, such as an identifier, operator or punctuation mark. If not, the READ CHARACTER function is executed again until a symbol is recognized.

The syntax analyser of the compiler then determines if this symbol conforms to CORRECT SYNTAX in accordance with the grammar of the programming language. If the symbol does not conform to the syntax an ERROR MESSAGE is displayed.

If the syntax is correct, the READ CHARACTER function is repeated until an error is found or the END OF PROGRAM is reached. In this event the CODE GENERATOR may be invoked if this function is performed as a separate pass. Alternatively, code generation may be performed concurrently with the lexical and syntactic analyses. The generated code may then be saved on disk and/or executed, at the option of the programmer, as indicated by the legend SAVE/EXECUTE OBJECT CODE.

For clarity in illustration it will be shown how the simple and widely-published compiler PL/0 of Prof. N. Wirth (Ref. 12) may be modified for implementation in the present invention. In the description below the following identifiers have been added and do not appear in the original PL/0 compiler as published: CONT, PEEK, RECOMPILE, PTR, PM and SP.

The first statement in the modified compiler is:

```
IF NOT CONT THEN
```

The boolean variable CONT is FALSE upon initial entry into the compiler signifying that this is not a continuation of a previous execution. That is, the compiler has just been entered for the first time during the present session. The subsequent assignment statements are therefore executed to initialize the contents of the arrays WORD, WSYM, SSYM, MNEMONIC, DECLBEGSYS, STATBEGSYS and FACBEGSYS shown at Pages 346, 347 of the Wirth treatise (Ref. 12). The values of these arrays remain fixed throughout execution of the compiler and the above conditional IF statement obviates the need to re-execute all of these assignment statements upon subsequent re-initializations of the compiler for recompilations. That is, after the first test of the variable CONT it is set equal to TRUE so as to bypass the assignment statements thereafter when recompilation is required.

After the conditional block of array assignments a PEEK assembly language function is invoked to read the content of the memory location immediately preceding the start of the source code buffer, in which location is stored the recompile flag. If this location has had the ASCII code for the letter 'R' stored therein by the editor then a compiler procedure RECOMPILE is invoked to reinitialize the variables ERR, CC, CX and LL, and to assign the value of constant AL (10) to the variable KK.

The RECOMPILE procedure also sets the value of a pointer variable PTR equal to the address of the beginning of the source code buffer. The pointer PTR is the memory pointer of the compiler's lexical analyser and is successively advanced from byte to byte of the source code to read the latter. The lexical analyser reads in the byte in the memory location pointed to by the pointer PTR.

The lexical analyser embodies another major change in the PL/0 compiler. It is embodied in the procedure GETSYM which also has nested therein the procedure GETCH.

GETSYM has as its first statement:

```
WHILE  CH = ' ' DO GETCH;
```

This constitutes part of an infinite loop which repeats for as long as the procedure GETCH returns the ASCII code (32) for a space. As explained below, the procedure GETCH will return the space code 32 whenever it reads the Pause Mark.

GETCH has as its first statement:

```
CH := PTR^;
```

so as to read into the variable CH the contents of the source memory location pointed to by the pointer variable PTR. The next statement of GETCH is:

IF CH = CHR(PM) THEN

thereby testing if the byte read is the Pause Mark PM which is a constant equal to 35. This value was chosen because it is a visible character and was otherwise unused.

Following the IF clause is the assignment:

CH := CHR(SP)

where SP is equal to the ASCII code (32) for a space. Control then returns to GETSYM where the condition of the WHILE clause is satisfied so that it again invokes GETCH. This sequence is repeated and results in an infinite loop for as long as the byte in the memory location being read is the Pause Mark.

After the editor changes that byte from the Pause Mark to the ASCII code for a space the loop will be broken because the boolean condition of the IF clause will no longer be satisfied (variable CH will no longer equal PM). Instead the following ELSE clause will be executed so as to advance the source memory pointer PTR to the next memory location by the statement:

PTR := PTR + 1 ;

whereafter the next invocation of GETCH will read the next source memory location to enable the compiler to continue its advance through the source code. The pointer PTR is repeatedly advanced with each successive call of GETCH until it reaches the new Pause Mark inserted by the editor, as described below.

Referring to Fig. 5, there is shown the sequence of operations of the editor. The INPUT CHARACTER function is first performed in response to the KEYSTROKE (Fig. 3). The editor then determines if the input byte is a CONTROL CHARACTER. If not, the character is entered into the source code buffer as indicated at CHAR INTO MEM BUFFER. The input character is also displayed on the screen as indicated at CHAR TO VIDEO CONSOLE. If the input

character is a control character, the editor then determines if it is a CARRIAGE RETURN? If not, the appropriate one of the editor's routines for handling control characters is called, as indicated by the legend TO CONTROL CHAR ROUTINE, and as described below with reference to Fig. 6.

Still referring to Fig. 5, if the input character is a carriage return then a new Pause Mark is written into the source code buffer adjacent the end of the current line, as indicated at INSERT NEW PAUSE MARK. The old Pause Mark is changed to a blank (space), as indicated by the legend REMOVE OLD PAUSE MARK.

For convenience in finding the location of the Pause Mark subsequently, a memory word location is reserved as a Pause Register for storage of the location address of the Pause Mark. The address of the new Pause Mark is thus stored in this memory register, as indicated by the legend UPDATE PAUSE REGISTER. The ASCII code for a carriage return (13) is then entered into the source code buffer adjacent the Pause Mark, as indicated by INSERT CR INTO MEM BUFFER. The ASCII code for a line feed (10) may be entered after the carriage return if this convention is desired.

Referring now to Fig. 6, there is shown the sequence of operations of the editor routines for handling control characters input at the console. The input character is first tested to determine if it is the code for the CURSOR UP operation. If not, it is then tested to determine if it is the code for the SCREEN UP operation. If not, the input control character is handled in a conventional manner which will not be further described, as indicated by the legend PROCESS OTHER CONTROL CHAR.

If the input control character is the code for the CURSOR UP

or SCREEN UP then the respective operation MOVE CURSOR UP or SCROLL SCREEN is performed. In the former case the cursor is moved up one line on the video screen. In the latter case the screen is erased and is rewritten to display those lines of the source code buffer immediately preceding the erased lines.

As indicated at INSERT NEW PAUSE MARK, a new Pause Mark is inserted adjacent the end of the source code buffer line immediately preceding the line now bearing the new cursor position. The operations REMOVE OLD PAUSE MARK and UPDATE PAUSE REGISTER are then performed in the same manner as described above with respect to Fig. 5.

The operation SET RECOMPILE FLAG causes reinitialization of the compiler when the latter resumes control of the CPU after return from the interrupt service routine. This flag is preferably a memory location wherein a predetermined code may be stored to inform the compiler that recompilation of the source code is required.

In the preferred embodiment this recompile flag is set to require recompilation whenever the cursor is moved up or the screen frame is scrolled up. That is, it is assumed that whenever the cursor is moved to point to source code which may have already been compiled that this code will be changed so as to require recompilation.

An alternative method would be to set the recompile flag only if the previously compiled code is actually changed, since it is possible that the programmer may scroll the screen up and then scroll down again without making any change in the source code.

Another alternative would be to maintain a memory register holding the address of the latest position of the compiler

pointer. The editor might then compare this address with that of the source location pointed to by the cursor to determine if the editing changes are being made to source code which has already been compiled.

Although these alternative schemes result in fewer recompilations, the preferred embodiment has the advantage of simpler implementation. Furthermore, the compilation process is so much faster than the manual typing of source code at the console that the compiler will recompile all but the largest programs and catch up with the programmer before the programmer can type more than a few new lines of code. Therefore the reduction of the number of recompilations to the absolute minimum is not essential.

The editor is written in Pascal with calls to sixteen external assembly language procedures and functions. Those routines unique to the present invention will now be described.

Upon entry to the editor the boolean variable ECONT is tested to determine if this invocation of the editor is the first entry of the present session or a continuation. If ECONT is FALSE then it is set equal to TRUE and the following procedures are called: INIT, NEWFILE, VECTOR and TOPLO.

The procedure INIT clears the source code buffer, sets the memory pointer to the start of the buffer, inserts the Pause Mark at the first location of the buffer, sets the contents of the Pause Register to the address of this first location, initializes the cursor to the first row and first column of the screen, and sets the recompile flag pointer to the memory location preceding the first byte of the buffer.

The procedure NEWFILE prompts the programmer to select

either a new file for entry of source code or an old file for editing. If the latter, the file is read into the source code buffer from a disk and the first screen of source code is displayed.

The procedure VECTOR calls the external assembly procedure POKE three times to store in low memory (20H) the jump vector to a subroutine SAVREGS which stores the contents of the CPU registers. In response to an interrupt activated by a keystroke at the console the CPU executes the RST4 call instruction and executes this jump vector and then the SAVREGS subroutine. After the registers are saved a jump instruction in the subroutine sends the CPU to the editor.

The TOPL0 procedure passes control to the PL/0 compiler. It is usually called after the editor has completed the character entry or editing function corresponding to the key struck at the terminal. In this case the procedure is called after initialization of the editor.

The next statement of the editor is not reached until an interrupt occurs in response to a keystroke. This statement reads into a variable the ASCII code input from the UART's received data register. This input byte is tested to determine if it is a control character or alphanumeric character (greater than 31). If the latter it is entered into the source code buffer and displayed on the video screen in the conventional manner.

If the input byte is either the control code for moving the cursor down or for scrolling the screen frame down, the appropriate procedure is called and concludes with an invocation of the procedure UPDATE. The latter enters into the source buffer a new Pause Mark adjacent the end of the old line in the

case of a cursor down operation, and adjacent the end of the invisible line preceding the first displayed line in the case of a scroll down operation. The procedure UPDATE also removes the old Pause Mark by substituting the ASCII code for a space (32) in place of the Pause Mark in the old location of the latter. The Pause Register is also updated to the address of the new Pause Mark location.

If the input byte is either the control code for moving the cursor up or for scrolling the screen frame up toward the beginning of the source, the corresponding procedure is called to perform the respective operation. This procedure concludes with invocations of the previously described procedure UPDATE and also the procedure RECOMPILE. The latter stores the ASCII code for the letter 'R' in the memory location immediately preceding the start of the source code buffer so as to constitute the recompile flag noted above. Upon completion of the cursor up operation or the screen scroll up operation the CPU will return to the compiler which will test the recompile flag, determine that the flag is set, and then call its reinitialization procedure to force the compiler to recompile the source code from the beginning of the source buffer.

If the input byte is the ASCII code for a carriage return (13) the procedure CRET is called. This routine enters into the source buffer a new Pause Mark adjacent the carriage return code, removes the old Pause Mark, and updates the Pause Register, among other more conventional functions such as adding a line feed code to the buffer, updating the cursor, and scrolling the video display if the present line is the last line of the screen.

It should be understood that the preferred embodiment de-

scribed above and shown in the drawings is merely illustrative of one of the many forms which the invention may take in practise, and that numerous modifications thereof may be made by those skilled in the art without departing from the scope of the invention as defined in the appended claims.

For example, instead of the Pause Mark implemented as a predetermined code entered into a memory location within the source code buffer, the pause location may be defined for the compiler by a memory address stored in a register. The compiler may then be prevented from analysing code stored in memory locations beyond this address which may be incremented and decremented by the editor.

Furthermore, the interrupt which causes control of the CPU to pass from the compiler to the editor may be activated by a timer or clock instead of by the keyboard. That is, the compiler may be periodically interrupted and the input port polled to test if a key has been struck. If not, the interrupt is terminated and control returns to the compiler. If polling the port reveals that a key has been struck then the interrupt service routine editor takes control and is executed in the manner described above for the disclosed preferred embodiment. For most applications clock interrupts at intervals of about every 10 to 30 milliseconds should be frequent enough to keep up with keys stroked at the keyboard.

Furthermore, the recompile flag may be set whenever the compiler determines that the source code contains an error. That is, it may be assumed that whenever an error is revealed the source code will be changed so as to require recompilation.

Another possible modification is to eliminate the

requirement of recompilation from the very beginning of the source code in those instances where the error occurs in the last completed line of source code. During the compilation of the line the resulting register values, table entries, stack manipulations, variable assignments and code buffer entries are temporarily stored and are not finally entered until the syntax analysis of the source line is completed and determines that the line conforms to the grammar. If the line contains an error these temporary entries are discarded and the compiler pointer is moved back to the end of the previous line, thereby obviating recompilation. However, this scheme will still require recompilation if source lines previous to the last line are modified.

Still another possibility would be to have the editor advance the Pause Mark after entry of each character or after entry of each delimited symbol. This would have the advantage of revealing an error almost instantly upon its entry at the keyboard, instead of waiting until completion of the current line. The disadvantage would be that recompilation would be required for every minor typing error without giving the programmer a chance to correct it before it is scanned and parsed.

BIBLIOGRAPHY

1. L. V. Atkinson, et al., "Context Sensitive Editing as an Approach to Incremental Compilation", The Computer Journal, Vol. 24, No. 3, 1981, pp. 222-229, Heyden & Son Ltd.

2. L. V. Atkinson and S. D. North, "COPAS--A Conversational Pascal System", Software--Practice and Experience, Vol. 11, 1981, pp. 819-829, John Wiley & Sons Ltd.

3. R. B. Ayres and R. L. Derrenbacher, "Partial Recompilation", Spring Joint Computer Conference, 1971, pp. 497-502.

4. F. T. Baker and H. D. Mills, "Chief Programmer Teams", Classics in Software Engineering, Edited by E. N. Yourdon, Yourdon Press, pp. 195-204.

5. L. Bolliet, et al., "DIAMAG: A Multi-Access System for On-Line Algol Programming", Proc.--Spring JCC 1967, pp. 547-552.

6. P. J. Brown, "Writing Interactive Compilers and Interpreters, Ch. 2.4, pp. 41-45, John Wiley & Sons.

7. J. Earley and P. Caizergues, "A Method for Incrementally Compiling Languages with Nested Statement Structure", Communications of the ACM, Vol. 15, No. 12, Dec. 1972, pp. 1040-1044.

8. K. Lock, "Structuring Programs for Multiprogram Time-Sharing On-Line Applications", Proc.--Fall JCC 1965, pp. 457-472.

9. J. Pournelle, User's Column, BYTE, Sept. 1982, p. 324.

10. J. L. Ryan, et al., "A Conversational System for Incremental Compilation and Execution in a Time-Sharing Environment",

Proceedings--Fall Joint Computer Conference, 1966, pp. 1-21.

11. S. Sonderstrup and M. Pelle, "COMAL-80 A New Language?", Dr. Dobb's Journal, No. 56, June 1981, pp. 14-15, 46-47.

12. N. Wirth, "Algorithms + Data structures = Programs", Ch. 5, pp. 280-347, 1976, Prentice-Hall, Inc.